ISSN 1673-9418 CODEN JKYTA8
Journal of Frontiers of Computer Science and Technology
1673-9418/2010/04(10)-0927-13
DOI: 10.3778/j.issn.1673-9418.2010.10.007

E-mail: fcst@vip.163.com
http://www.ceaj.org
Tel: +86-10-51616056

MESHJOIN*：实时数据仓库环境下的数据流更新算法*

林子雨+，林 琛，冯少荣，张东站
厦门大学 计算机科学系，福建 厦门 361005

MESHJOIN*: An Algorithm Supporting Streaming Updates in a Real-time Data Warehouse*

LIN Ziyu+, LIN Chen, FENG Shaorong, ZHANG Dongzhan

Department of Computer Science, Xiamen University, Xiamen, Fujian 361005, China

+ Corresponding author: E-mail: ziyulin@xmu.edu.cn

LIN Ziyu, LIN Chen, FENG Shaorong, et al. MESHJOIN*: An algorithm supporting streaming updates in a real-time data warehouse. Journal of Frontiers of Computer Science and Technology, 2010, 4(10): 927-939.

Abstract: A new algorithm called MESHJOIN* is proposed to support streaming updates under real-time data warehouse environment. It has the following distinct features: (1) Relation R is organized in blocks and hashes so as to avoid the reading of unusable tuples for the current join operation as much as possible, through which the amount of tuples involved in a join is much reduced, thus enhancing the efficiency of the join operation; (2) Multi-thread parallel execution technology is adopted here, and the order of read operation and join operation is optimized according to engineering theory so as to maximize the efficiency of join algorithm; (3) Reasonable scheduling of real-time tuples and near-real-time tuples is achieved according to the relationship between the current system service rate and the tuples arriving rate, so that the requirement for the processing of real-time tuples is satisfied. Experimental results show that MESHJOIN* can achieve much better performance than MESHJOIN.

Key words: data warehouse; streaming update; join

摘 要：提出了一种新的实时数据仓库环境下的数据流更新算法——MESHJOIN*算法。算法的特性有：(1) 关系R采用了分块和散列的组织形式，尽可能避免对当前连接无效元组的读取，减少连接操作所涉及元组

*The National Natural Science Foundation of China under Grant No.50604012 (国家自然科学基金).
Received 2010-05, Accepted 2010-07.

的数量, 从而提高连接算法的效率; (2) 采用了多线程并发连接技术, 并根据工程学原理, 实现了连接操作和关系 R 读取操作的最佳调度, 保证了连接算法效率的最大化; (3) 根据当前系统的服务率和数据流元组的到达率之间的关系, 合理调度实时元组和准实时元组的执行, 保证了系统对实时元组的处理要求。实验结果表明, MESHJOIN*算法可以取得比 MESHJOIN 算法更好的性能。

关键词: 数据仓库; 数据流更新; 连接
文献标识码：A 中图分类号：TP311

1 引言

实时数据仓库[1–4]已经在越来越多的企业中得到应用, 并且产生了很好的经济效益。在实时数据仓库的 ETL(extact, transform, load)过程中, 有一种常见的数据转换操作, 即 $S\infty_C R$, 其中, S 表示更新语句对应的关系表, R 表示数据仓库中的关系表, C 表示特定的连接操作。比如等值连接, 为了方便论述, 本文都默认采用等值连接, 扩展到其他情形是不难做到的。在数据流式更新中, S 包含了源源不断到达的元组, 其内容不断发生变化, 每个元组都要完成和 R 的连接操作。R 则驻留在数据仓库中, 内容相对稳定。R 通常包含大量元组, 占用比较大的存储空间。连接操作是在内存中执行的, 由于内存大小的限制, 在某个时刻 R 中只有一部分元组可以放入内存。这时就会面临一个难题, 即在执行 S 和 R 的连接操作时, S 中的元组以数据流的方式不断地快速到达, 而关系 R 中的元组从磁盘读入内存, 由于需要一定的 I/O 开销, 因此, 速度相对较慢。这就很可能导致连接操作的处理速度落后于 S 中元组的到达速度, 继而造成连接操作的延迟处理, 影响 ETL 的实时性。

如上所述, 由于内存的限制和连接所涉及关系的元组的到达速度差, 在实时数据仓库环境下的连接操作 $S\infty_C R$ 与传统数据仓库环境下的连接操作有着很大不同。因此, 在传统数据仓库环境下能够发挥良好性能的连接算法, 比如索引嵌套循环连接、归并连接和散列连接, 将无法直接应用到实时数据仓库环境中。以索引嵌套循环连接为例, 由于实时数据仓库的实时分区中的数据通常不存在索引, 这种方法就无法直接应用。文献[5]的研究发现, 即使存在索引, 使用 S 中的更新元组 s 对关系 R 的索引进行查找时, 也需要很高的随机读取开销, 这导致连接操作处理速度通常无法赶上数据流元组的到达速度。

本文提出了一种实时数据仓库环境下的数据流更新算法——MESHJOIN*算法, 它是对文献[5]提出的 MESHJOIN 算法的改进算法。与原算法相比, MESHJOIN*算法针对关系 R 采用了分块和散列的存储形式, 可以尽可能避免对当前连接无效元组的读取, 减少连接操作所涉及元组的数量, 从而提高连接算法的效率。另外, MESHJOIN*算法采用了多线程并发连接技术, 并根据工程学原理, 实现了连接操作和关系 R 读取操作的最佳调度, 保证连接算法效率的最大化。在 MESHJOIN*算法中, 当数据流更新元组中同时存在实时元组和准实时元组时, 优化器可以根据当前系统的服务率和数据流元组的到达率之间的关系, 合理调度不同类型元组的执行, 保证系统对实时元组的处理要求, 并在此前提下尽可能提高系统整体处理效率。实验表明, MESHJOIN*算法可以取得比 MESHJOIN 算法更好的性能。

本文第 2 章介绍了 MESHJOIN 算法; 第 3 章介绍 MESHJOIN*算法的朴素形式; 第4章介绍了 MESHJOIN*算法的高级形式; 第 5 章是实验设计及实验结果; 相关工作在第 6 章进行介绍; 最后,

对全文进行总结。

2 MESHJOIN 算法

由于数据流更新算法 MESHJOIN*是建立在 MESHJOIN 算法[5]基础之上的，因此，首先简要介绍一下 MESHJOIN 算法。如图 1 所示，数据流 S 和关系 R 分别对应前面论述的典型连接操作中的关系，并且符合对这两种关系的特点的描述。MESHJOIN 方法会对两种输入不断执行连接操作并产生结果输出。由于内存大小的限制，在每个特定的滑动窗口内，数据流 S 中只能有 w 个元组放入内存(用 S_w 表示这些元组)，关系 R 中也只能有 b 页数据放入内存(用 R_w 表示这些元组)。对一个较长的时期而言，数据流 S 中的每个元组 t 只被读取一次，在完成与 R 中的所有元组的连接操作以后即被丢弃(因为这时与 t 相关的所有结果都已经生成)，然后下一个元组进入滑动窗口(内存)；而关系 R 中的元组则会被循环往复地读入内存，被多次反复使用。通过这种方式，读取 b 页关系 R 的数据的开销就可以被分摊到 wN_R/b 个当前滑动窗口中的元组，这就有效平衡了数据流元组到达速度与关系 R 元组读取速度之间的差异。

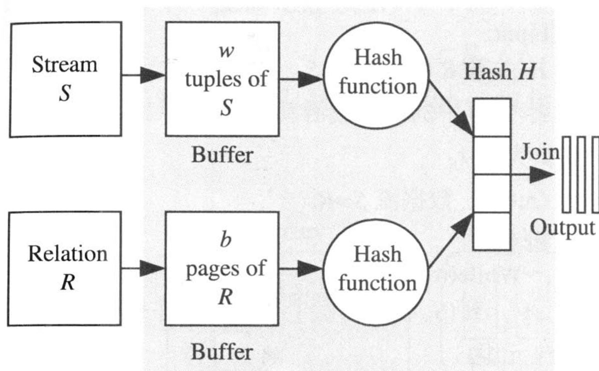

Fig.1 The description of MESHOJOIN algorithm

图 1 MESHJOIN 算法示意图

但是，经过仔细分析和实验发现，MESHJOIN 算法存在一个明显的不足，即需要读取关系 R 中的全部元组。假设关系 R 包含 N_R 页数据，并且 N_R 能够被 b 整除，即 $k=N_R/b$。在 MESHJOIN 算法中，每个滑动窗口内，有 b 页关系 R 的数据被读入内存，在经过 N_R/b 次以后，关系 R 中的所有元组将被完整地读取一遍，虽然这些元组是被分批次读取的。通过实验发现，在每次被读取的 b 页数据所包含的元组中，并非所有元组都参与到了连接操作中，甚至在一些时候，只有很少元组是参与连接操作需要的。关系 R 中这些“无效元组”的读取，需要不小的 I/O 代价，浪费了对于实时 ETL 而言宝贵的时间。在每个滑动窗口内，如果尽可能只把“关系 R 中那些可以与 S_w(如前文所述，S_w 表示位于当前滑动窗口内的元组)进行连接的元组”读入内存，而忽略与当前连接操作无关的元组，那么，就可以大量减少 I/O 代价，提高连接操作性能。由于在 MESHJOIN 算法中，关系 R 中的元组会被多次循环读取，因此，这种 I/O 代价的节省具有累积性。本文的 MESHJOIN*算法可以有效弥补这一不足，显著优化连接操作的性能。

3 MESHJOIN*朴素算法

在 MESHJOIN 算法基础上，提出了 MESHJOIN*算法，除了继承 MESHJOIN 算法已有的优点以外，MESHJOIN*算法还具有自身的优势。MESHJOIN*支持高效的数据流更新，可以对快速的数据流 S 和大尺寸的数据仓库关系 R 执行高效的连接操作，这主要得益于以下技术特性：

(1) 通过顺序扫描实现对关系 R 的快速访问；

(2) 尽可能只读取关系 R 中的与当前连接操作相关的元组，大大降低了 I/O 开销；

(3) 把 I/O 开销进一步分摊给大量的数据流元组；

(4) 可以根据连接操作性能要求，优化关系 R 中元组读入内存的顺序；

(5) 实现了实时更新元组和准实时更新元组的合理调度。

其中，第(2)、(4)、(5)这 3 个特性是原来的 MESHJOIN 算法所没有的。为了更好地理解 MESHJOIN*算法，先介绍该算法的朴素版本，即

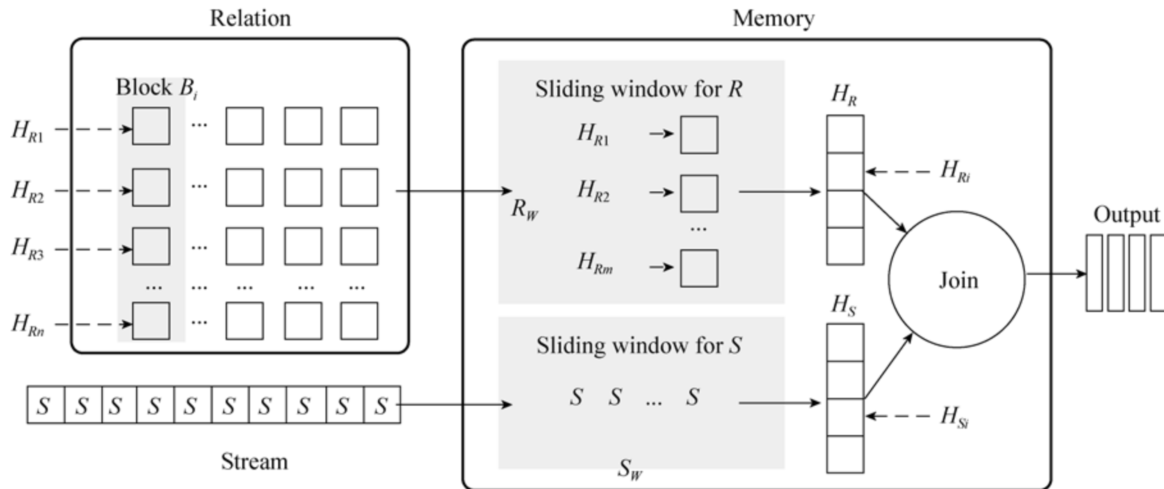

Fig.2 The description of naïve MESHJOIN* algorithm
图 **2** **MESHJOIN***朴素算法示意图

算法特性只包含上述 5 个特性的前 3 个。在理解了朴素的 MESHJOIN*算法后，将介绍更加复杂的高级 MESHJOIN*算法，它包含了所有 5 个特性。

3.1 MESHJOIN*朴素算法描述

如图 2 所示，在 MESHJOIN*朴素算法中，关系 R 被分割成多个块，每个块 B_j 又采用散列函数分割成多个散列桶 $H_{R1},H_{R2},\ldots,H_{Rn}$。在 MESHJOIN 算法中，关系 R 的每个块会被完整读取；而在 MESHJOIN* 朴素算法中则避免了这个问题，MESHJOIN*朴素算法只读取那些“与 S_w 可能具有连接关系的元组”，从而减少 I/O 开销。这个问题采用散列连接方法是很容易解决的。散列连接算法的基本思想就是：如果关系 R 的一个元组与关系 S 的一个元组满足连接条件，那么它们在连接属性上具有相同的值。经过散列函数映射以后，关系 R 的那个元组和关系 S 中的那个元组一定分别位于散列桶 H_{Ri} 和 H_{Si} 中。这就意味着，H_{Ri} 中的元组只需要与和H_{Si}中的元组相比较，而没有必要与 S 的其他散列桶中的元组比较。因此，采用散列函数(与关系R的映射采用相同的散列函数)把 S_w 映射到散列表 H_S 中，然后，在读取关系 R 的某个块 B_i 时，只有当散列表 H_S 中的散列桶 H_{Si} 存在元组时，才把块 B_i 中的散列桶 H_{Ri} 读入内存，而不是把 B_i 中的所有散列桶都读入内存。关系 R 的元组被读入内存以后，被存放在散列表 H_R 的相应位置，数据流 S 的元组被读入内存以后，内存中的这些数据流元组 S_w 被散列函数映射到散列表 H_S 中，连接算法完成对散列表 H_R 和 H_S 中元组的连接操作并输出结果。随着数据流的不断到达，关系 R 会被循环往复地读进内存，从而源源不断地输出连接结果。

算法 **1** MESHJOIN*朴素算法

Input:

1. 关系 R 和数据流 S

2. 内存中的数据流滑动窗口大小 λ_s 和关系 R 滑动窗口大小 λ_R

Output: 数据流 $S\infty R$;

Begin

While(true)

If (S_w 中的元组个数小于 λ_s 并且存在新到达数据流元组)

从数据流 S 中读取元组 s 到数据流滑动窗口内;

把元组 s 映射到散列表 H_S;

End if

根据 H_S 中的元组分布情况，从关系 R 中顺序读取一定数量的元组到关系 R 滑动窗口 R_w;

把 R_w 的元组存放在散列表 H_R 相应的位置;

For each (R_w中的元组 r)

根据散列连接方法输出 $r\infty H_S$;

End for

If (S_w中的元组 s 已经与关系 R 的所有可能元组都已经执行过连接操作)

从 S_w中删除元组 s;

End if

End while

End

算法 1 显示了 MESHJOIN*朴素算法的基本过程。在该过程中，需要根据 H_S中的元组分布情况，从关系 R 中顺序读取一定数量的元组到关系 R 滑动窗口 R_w。前面已论述，这些被读取的关系 R 的元组必须是那些“与 S_w可能具有连接关系的元组”，这里称这些元组为“关系 R 的当前连接有效元组”，或简称为“有效元组”。现在用更加形式化的方式定义关系 R 的当前连接有效元组的确定准则。

定义 1 (关系 R 的当前连接有效元组确定准则) 已知数据流滑动窗口中的元组集合 S_w，根据散列函数 h 映射得到的散列表为 H_S, H_{Si}表示散列表 H_S的第 i 个桶，t_{Si}表示 H_{Si}中的元组个数；磁盘中的关系 R 被分割成多个块，每个块 B_j又采用散列函数 h 分割成多个散列桶 $H_{R1},H_{R2},\ldots,H_{Rn}$。则关系 R 的块 B_j中可能满足连接条件(与 S_w中的元组连接)的元组集合为 $B_j^* = \{H_{Ri} \mid H_{Ri} \in B_j \wedge t_{Si} \neq 0\}$，关系 R 中所有可能满足连接条件的元组集合为 $R^* = \bigcup B_j^*$。

关系 R 中的元组是按照块为单位被顺序读取的。对于某个元组 s，假设当它进入数据流滑动窗口时，关系 R 的第一个块 B_0(确切地说，是块中符合条件的元组)正被读入内存，完成连接操作以后，B_0被从内存中删除，下一个块 B_1被读入内存，依次类推；那么，当块 B_0 再一次被读入内存时，元组 s 就完成了与关系 R 的所有元组的连接操作，该元组就可以从数据流滑动窗口中删除，让下一个数据流元组进入数据流滑动窗口。

为了更好地理解 MESHJOIN*朴素算法，给了一个简单的例子。

例 1 如图 3 所示，磁盘关系 R 包含两个块 B_0和 B_1，数据流包含多个元组 $s_1,s_2,\ldots$。在内存中，属于关系 R 的滑动窗口只能容纳一个块，属于数据流的滑动窗口只能容纳两个元组。下面是连接操作的具体过程：

(1) 在 $T=T_0$时刻，数据流元组 s_1进入数据流滑动窗口，当前将要被读取的关系R 的块是 B_0，根据“关系 R 的当前连接有效元组的确定准则”，可以由块 B_0 得到它的当前连接有效元组 B_0^*，然后对 B_0^*和 s_1执行散列连接；

(2) 在 $T=T_1$时刻，数据流元组 s_2进入数据流滑动窗口，这时数据流滑动窗口已经达到所能容纳的最大元组值(两个元组)。当前将要被读取的关系 R 的块是 B_1，根据“关系 R 的当前连接有效

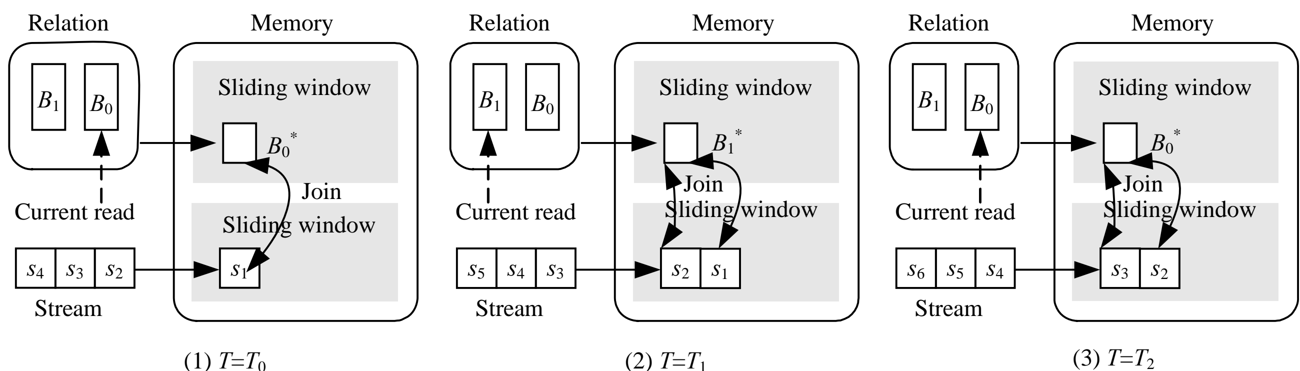

Fig.3 An example of naïve MESHJOIN* algorithm

图 3 一个 **MESHJOIN***朴素算法实例图

元组的确定准则”，可以由块 B_1 得到它的当前连接有效元组 B_1^*，然后对 B_1^* 和 s_1 及 s_2 执行散列连接；这时，s_1 已经执行完和关系 R 的所有块 B_0 和 B_1 的连接操作，因此，可以输出 $s_1 \infty R$ 的结果，并把 s_1 从数据里滑动窗口中删除；

(3) 在 $T=T_2$ 时刻，数据流元组 s_3 进入数据流滑动窗口，当前将要被读取的关系 R 的块是 B_0，又一个新的周期开始。

现在继续以上面的例子为基础，说明MESHJOIN*算法相对于 MESHJOIN 算法而言是如何节省 I/O 代价的。假设关系 R 的每个块都被散列函数 h 均匀地分成 5 个桶 $H_{R0},H_{R1},\ldots,H_{R4}$，这 5 个散列桶的入口地址 i 分别是 0,1,2,3,4，每个桶都包含 10 个元组。在 $T=T_0$ 时刻，当数据流元组 s_1 进入数据流滑动窗口时，使用散列函数 h 映射得到它在散列表中的入口地址为 2，那么，根据“关系 R 的当前连接有效元组的确定准则”，当前被读取的块 B_0 中，只有 H_{R2} 是当前连接有效元组，会被读入内存，即 I/O 开销只相当于 10 个元组。而 MESHJOIN 算法则需要读取块 B_0 的所有 50 个元组，因此，MESHJOIN*算法比 MESHJOIN 算法节省了相当于 40 个元组的开销。

3.2 关系 *R* 有效元组的读取

对于关系 R 的各个块 B_j 而言，不同的 B_j^* 所包含元组的数量可能差别较大。这既与关系 R 所采用的散列函数的特性有关，也和当前的 S_w 被映射到散列表中的分布有关。当可以采用有效的散列方法保证关系 R 的元组在不同散列桶中的均匀分布时，那么当前的 S_w 被映射到散列表 H_S 中的入口地址分布就成了影响“B_j^* 所包含元组的数量”的最主要因素。

在 MESHJOIN*朴素算法描述中，为了简化问题，假设每次都以块为单位把磁盘关系 R 的元组读到内存。但是，实际上，由于 B_j^* 所包含元组的数量的差异，某些块 B_j 的有效元组 B_j^* 可能比较少，如果每次连接操作只读取 B_j^*，则会出现内存中的关系 R 滑动窗口出现比较多的剩余空间，导致严重的空间浪费。另外，磁盘数据读取 I/O 开销可以主要分为两个部分：一部分是读取数据的开销，和数据的量成正比；另一部分是磁盘启动和结束开销，这部分开销相对比较固定。当每次从磁盘中读取比较多的数据时，固定部分的开销就可以被这些数据分摊，一次读取数据越多，单位数据的 I/O 总开销就越少。因此，从这个角度而言，也有必要在每次连接操作中，读取尽量多的关系 R 的元组进入内存。

这里采用的方法是：首先，在块的划分上，保证内存中的关系 R 滑动窗口 W_R 能够至少容纳一个完整的块 B_j，也就是说，即使块 B_j 中的所有元组都是当前连接的有效元组(即$B_j=B_j^*$)，W_R 也能容纳所有这些有效元组，即 $\forall j(V_{B,j} < V_{W_R})$，其中 $V_{B,j}$ 表示块 B_j 的存储空间，V_{W_R} 表示 W_R 的空间大小；其次，当 B_j^* 的元组读入 W_R 以后，如果 W_R 剩余的空间还够容纳 B_{j+1}^*，即有下式子成立：

$$V_{B,i} + V_{B,i+1} + \ldots + V_{B,j} + V_{B,j+1} < V_{W_R}$$

那么继续读入 B_{j+1}^*，依次类推，直到 W_R 剩余的空间不够容纳下一个块的所有有效元组。

4 MESHJOIN*高级算法

文章开始部分，曾介绍了实时数据仓库的特点，即把数据分为实时数据和历史数据分别存放在实时分区和历史分区中。历史分区的数据每天更新一次，而实时分区的数据则进行实时更新，并且在一天结束的时候加载到历史分区。这就意味着实时分区中包含的数据大体上可以划分为两大类：实时数据和准实时数据，前者必须实时更新，而后者的更新周期介于实时和一天之间，比如每 2 小时更新一次。但是，现有的连接算法，以及 MESHJOIN*朴素算法中，并没有对这两种数据进行区分。实际上，把实时数据和准实时数据不加以区别地处理，会影响到连接算法的整体性能，尤其是系统比较繁忙的时候，宝贵的资源

应该尽量优先分配给实时数据，而对于准实时数据则可以选择适当的时机处理，比如实时数据到达低峰期。一种可以想到的方法是，把二者完全分开处理，彼此独立执行连接操作，但这又大大增加了系统总体开销。因此, MESHJOIN*高级算法，可以有效统筹和优化对两种不同类型数据的连接操作过程，在保证实时数据更新性能的同时，提高准实时数据的更新作业吞吐量，使系统总体性能最优化。

4.1 算法描述

如图 4 所示, MESHJOIN*高级算法在内存中为数据流元组设置了两个不同的滑动窗口，即实时(real-time)数据滑动窗口和准实时(near-real-time)数据滑动窗口，并用S_{w_real}和S_{w_near}分别表示位于这两个滑动窗口内的元组的集合。S_{w_real}和S_{w_near}中的元组将被散列函数 h 分别映射到散列H_{S_real}和H_{S_near}中，优化器 optimizer 根据散列H_{S_real}和H_{S_near}中元组的分布情况，给关系 R 读取器 relation reader 发出指令，命令它按照制定的顺序从磁盘关系 R 中读取元组，从而使得连接算法性能达到最优化。优化器 optimizer 的另一个任务是，监督数据流中实时元组(必须采用实时更新方式)的到达速率 μ_{arrive}(单位时间 t 内到达的实时元组的数目)和连接算法服务实时元组的速率 $\mu_{service}$(单位时间t内完成连接操作的实时元组的数目)，从而协调系统资源在实时元组和准实时元组之间的分配，实现系统整体性能最优化。

4.2 关系 *R* 元组的读取

为了提高连接算法的整体性能，这里采用多线程散列连接。如图 5 所示，在多线程散列连接中，线程 i 负责执行散列 H_S和 H_R的桶 H_{Si}和 H_{Ri}之间的连接，假设由此得到的连接结果是 r_i。不同线程得到的连接结果之间不具有交叉性，因此，可以直接把连接结果进行合并作为最后结果输出，即：

$$S\infty R = r_1 \cup r_1 \cup \ldots \cup r_n$$

对于当前滑动窗口内的连接操作，当 S_w 中的元组被映射到 H_S 以后，多线程并发连接操作开始执行。当采用多线程并发执行时，每个线程需要在读到自己所需要的数据以后，才能执行连接操作。对于线程 i 而言，需要的数据是 H_{Ri} 中的元组，令 C_{read_R} 表示从磁盘关系 R 读取元组到 H_{Ri}

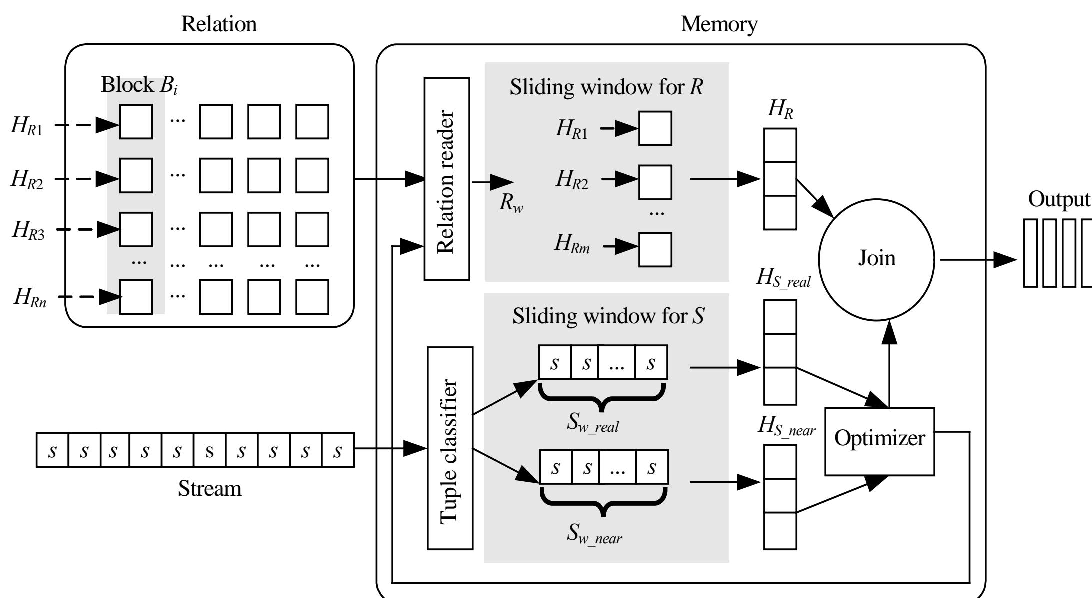

Fig.4 The description of advanced MESHJOIN* algorithm

图 4 **MESHJOIN***高级算法示意图

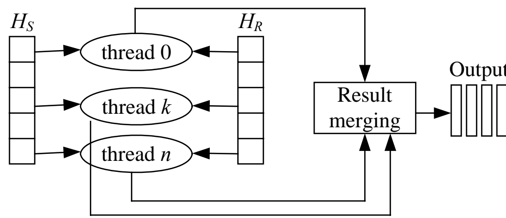

Fig.5 Multi-thread hash join
图 **5** 多线程散列连接图

中的代价, $C_{hashjoin}$ 表示 H_{Si} 和 H_{Ri} 中的元组进行连接的代价, 因此有：

$C_{join}=C_{read_R}+C_{hashjoin}$

假设线程 i 执行连接操作的时间开销为 $C_{join}(i)$, 由于多线程执行的并行性, 则当前滑动窗口内的连接操作 $S_w\,|\times|\,R_w$ 的时间开销 C_{join} 是所有线程中时间开销的最大值, 即 $C_{join}=\max(C_{join}(i))$。在多线程连接中, 不同线程读取关系 R 的操作是串行执行的, 无法并发执行, 由于并发读取会导致磁盘磁头在不同区域来回移动, 大大增加了读取操作的总体开销。因而, 让磁盘读取操作和散列连接采用并发执行, 即某个线程在执行读取操作时, 另一个线程可以并发执行散列连接操作。根据工程学原理, 为了使并行作业在最短时间内完成, 应该让执行时间比较长的作业优先执行。下面以一个例子加以说明。

对于不同的线程 i 而言, 由于假设关系 R 的元组均匀分布在各个块的各个桶中, 每个块的各个桶所包含的元组数目大致相同, 不同线程的 C_{read_R} 也大致相同, 因此不同线程的 $C_{join}(i)$主要区别在于 $C_{hashjoin}(i)$的不同。而在当前滑动窗口内, 数据流元组通常不会均匀地分布到散列 H_{S_real}和 H_{S_near}的各个桶中, 导致不同桶中包含的元组数量的差异, 从而影响 $C_{hashjoin}(i)$的大小。而 $C_{hashjoin}(i)$的值取决于散列桶中的元素数量, 根据上面论述, 为了最小化整体连接操作开销, 需要为包含元素数量比较多的桶优先执行读取和连接操作。

形式化地, 假设 $H_{S_real}(i)$和 $H_{S_near}(i)$分别是散列 H_{S_real} 和 H_{S_near} 的第 i 个桶, $0\le i\le m$, $n_a(i)$和 $n_b(i)$分别表示 H_{S_real} 和 H_{S_near} 的第 i 个桶中包含的元素个数, 令 $n_{total}(i)=n_a(i)+n_b(i)$, 并且有 $n_{total}(0)\ge n_{total}(1)\ge\ldots\ge n_{total}(m)$。假设采用 m 个线程并发连接中, 线程 i 执行连接操作 $H_{S_real}(i)\,|\times|\,R_w$ 和 $H_{S_near}(i)\,|\times|\,R_w$, 则能够使总体连接时间最小化的最优线程执行顺序是线程 $0,1,2,\ldots,m$。

4.3 实时元组和准实时元组的协调处理

在对实时元组和准实时元组进行区分时, MESHJOIN*算法优先处理实时元组, 在有多余资源的情况下再处理准实时元组。两种不同类型的元组的区分工作是由元组分类器 tuple classifier(见图 4)完成的, 这是比较容易实现的, 可以事先指定哪些表的数据需要实时更新, 哪些表的数据是准实时更新, 元组分类器可以根据数据流更新元组 s 所涉及的关系表的名称判断元组 s 的类型。随后, 元组 s 将根据其类型被放置在实时元组滑动窗口(这个窗口内的元组集合为 S_{w_real})和准实时元组滑动窗口(这个窗口内的元组集合为 S_{w_near})。优化器将根据数据流中实时元组到达速率 μ_{arrive} 和连接算法服务实时元组的速率 $\mu_{service}$, 来综合调度对两种不同类型元组的处理。当 $\mu_{arrive}>\alpha.\mu_{service}$ 时($0<\alpha<1$ 是一个服务性能保证系数), 算法需要停止对准实时元组的处理, 把所有资源都用于实时元组的处理; 当 $\mu_{arrive}\le\alpha.\mu_{service}$ 时, 算法可以把两种类型元组进行统一处理, 增加连接算法的总体吞吐量。这里需要指出的是, 当在 t_a 时刻出现 $\mu_{arrive}>\alpha.\mu_{service}$ 并停止对准实时元组的处理时, 需要记录一个处理断点, 也就是要记录下此刻算法正在读取的磁盘关系 R 的块的编号, 等到算法在未来某个时刻 t_b 再次读到这个块时, 优化器会检查 t_b 时刻 μ_{arrive} 和 $\alpha.\mu_{service}$ 的关系, 如果这时 $\mu_{arrive}>\alpha.\mu_{service}$, 则仍然继续停止对准实时元组的处理, 否则恢复对两种元组的统一处理。这样可以保证准实时元组不会遗漏与关系 R 中元组的连接操作。

5 实验设计与结果

本章介绍实验设计和实验结果。目的在于说明改进的算法 MESHJOIN*能够比原算法 MESHJOIN 取得更好的性能。

5.1 实验设计

数据集：实验采用来自中国某个移动通信运营商的业务数据，并在此基础上构建了关系 R 和更新数据流元组，每个都包含 600 万个元组。在构建关系 R 和更新数据流元组时，会考虑连接属性值的扭曲分布系数 *Zipf*[5]，当 *Zipf* 值为 0 时，各种不同连接属性值的元组的数量，在连接属性值域区间内均匀分布，随着 *Zipf* 值的增大，分布扭曲程度就越大。在下面论述中，用 *Zipf_R* 表示关系 R 的数据扭曲分布系数，用 *Zipf_S* 表示更新数据流 S 的数据扭曲分布系数。

实验环境：AMD Athlon 64 3200+ 2.20 GHz CPU，1 GB 内存，操作系统是 Windows XP Professional 2002。为了简化实验设置，数据源和数据仓库以及实时 ETL 操作全部在同一台 PC 上完成。

5.2 实验结果

实验 **1** MESHJOIN*算法和 MESHJOIN 算法以及 INL(indexed nested loops)算法的最大吞吐率。实验中，在连接算法执行过程中，关系 R 保持不变。令 m 表示内存空间大小与关系 R 大小的百分比，在实验过程中，把 m 从 1%逐渐增加到 10%，也就是把内存空间大小从 6 MB 增加到 60 MB，同时把 *Zipf_R* 和 *Zipf_S* 值都固定在 0.5，并观察 3 种算法的最大吞吐率的变化情况。从图 6 可以看出，MESHJOIN*算法和 MESHJOIN 算法的连接操作性能明显优于 INL 算法，同时，MESHJOIN*算法取得了比MESHJOIN 算法更大的最大吞吐率。而且，在不同的内存大小情况下，MESHJOIN*算法的最大吞吐率小幅度稳步增加。导致 MESHJOIN*算法性能改进的主要原因，就在于它只需要读取当前连接有效元组而不是全部元组。

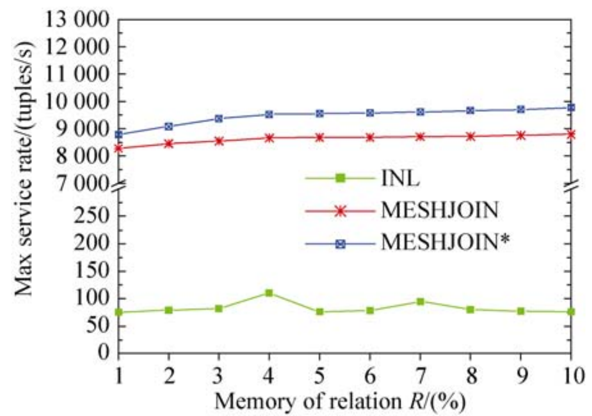

Fig.6 Maximum throughput of the three algorithms under different memory sizes

图 **6** 不同大小内存下 **3** 种算法的最大吞吐率

实验 **2** 在不同的 *Zipf_S* 值情况下 3 种算法的最大吞吐率变化情况。这里把 *Zipf_R* 值固定在 0.5，内存大小取值为关系 R 大小的 5%，让数据流更新元组的 *Zipf_S* 值从 0.1 变化到 1.0。对 MESHJOIN*算法采用多线程并发连接方法。从图 7 可以看出，*Zipf_S* 值对 INL 算法几乎没有影响，INL 算法主要由在关系 R 的索引中查找匹配元组的性能决定，当关系 R 的 *Zipf* 值固定在 0.5 时，查找的性能不会发生变化。而 MESHJOIN 算法的性能则随着 *Zipf_S* 值的加大而降低，因为，当更新数据流的扭曲分布程度加大时，散列 H_S 的某些桶就会包含很多元素，而另外一些桶则包含很少元素，针

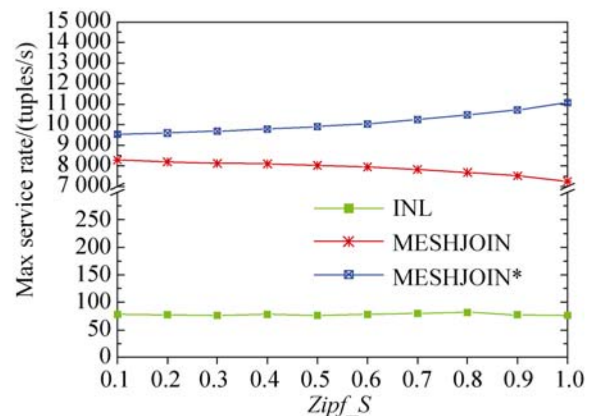

Fig.7 Maximum throughput of the three algorithms under different *Zipf_S*

图 **7** 不同 ***Zipf_S*** 值时 **3** 种算法的最大吞吐率

对散列中的元素的探查的总体时间开销就会增加，从而增加了连接操作时间。而 MESHJOIN*算法却可以有效利用数据分布的扭曲，利用合理安排不同关系 R 的块的读取顺序，实现连接速度最优化。从图 7 可以看出，更新数据流的数据扭曲程度越大，MESHJOIN*算法相对于 MESHJOIN 算法的性能优势就越明显。

实验 **3** 在不同的 *Zipf_R* 值情况下 3 种算法的最大吞吐率变化情况。实验基本设置与实验 2 基本相同，不同的是，实验 3 把 *Zipf_S* 的值固定在 0.5，令 *Zipf_R* 值从 0.1 变化到 1.0。从图 8 可以看出，*Zipf_R* 值的变化对 MESHJOIN*算法的性能几乎没有影响，由于该算法在进行多线程并发连接时的速度，取决于散列 H_S 中的元素分布，而已经把 *Zipf_S* 的值固定在 0.5，这就意味着散列 H_S 中的元素分布不变，因此，该算法的性能几乎不会发生变化。从图 8 同时可以看出，*Zipf_R* 值的变化对 MESHJOIN 算法和 INL 算法的性能产生了影响。当 *Zipf_R* 值越大时，关系 R 中元组在连接属性值上的分布扭曲程度越大，INL 算法访问关系 R 的 B+树索引代价就越大。MESHJOIN 算法性能下降的原因和实验 2 中的原因类似。

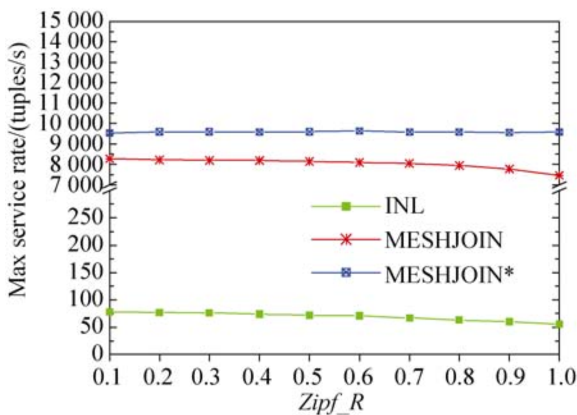

Fig.8 Maximum throughput of the three algorithms under different *Zipf_R*

图 **8** 不同 ***Zipf_R*** 值时 **3** 种算法的最大吞吐率

实验 **4** 针对实时与准实时数据流更新元组的综合调度性能。实验将验证在同时存在实时元组和准实时元组的情况下，MESHJOIN 算法和 MESHJOIN*算法的性能表现。令 MESHJOIN 算法的最大吞吐率为 u_{max}，并设置好相关的参数，即把 *Zipf_S* 和 *Zipf_R* 的值都固定在 0.5，内存空间大小与关系 R 大小的百分比 m 取为 10%。在此实验环境下，可以很容易测试出 u_{max} 的值。然后，令数据流更新元组到达的速率 f 从 $1.0*u_{max}$ 逐渐递增到 $2.0*u_{max}$，观察实时更新元组的延迟率(发生延迟的实时元组数量/实时元组的总数量)。这里令 r 表示数据流更新元组中实时元组与准实时元组的比例，图 9 和图 10 分别显示了 r=4 和 r=1/4 时的实时元组延迟情况。

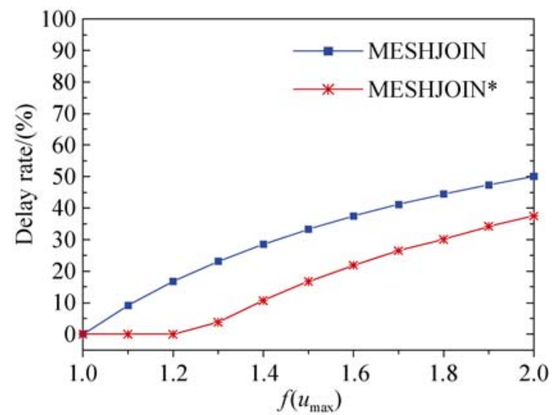

Fig.9 Latency of real-time tuples when r is 4

图 **9** ***r*=4** 时的实时元组延迟情况

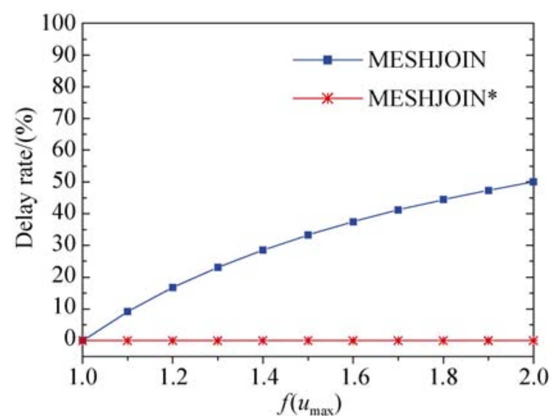

Fig.10 Latency of real-time tuples when r is 1/4

图 **10** ***r*=1/4** 时的实时元组延迟情况

从图 9 和图 10 中可以看出，当数据流到达速率大于连接操作算法处理能力时，在两种情况下，

MESHJOIN*算法都比 MESHJOIN 算法取得了更好的性能, 可以最大程度满足实时元组的需求。

6 相关工作

传统的数据仓库采用一次加载并周期更新的方法, 在进行数据更新时, 不允许进行分析操作。而实时数据仓库[1−4]必须支持实时的查询处理, 同时保证数据的实时性, 因此, 要求支持实时数据集成。为了最小化更新窗口, Labio[6]等人尝试使用批量加载工具来实现高性能的数据集成。2004 年, 相关文献中的某些方法以最小化整体更新工作量为出发点, 来确定更新数据仓库中的单个物化视图的最优策略。其他方法则将重点放在数据/表交换[7], 在这些方法中, ETL 工具获得很大的权限, 可以删除表、重加载表和操纵其他主要的数据库系统, 同时不影响终端用户的查询。文献[1, 8]则采用基于队列理论的实时数据集成方法。文献[8]为不同类型的 ETL 活动构建了相应的队列模型, 包括多输出队列模型、单输出队列模型以及 ETL 队列通用模型, 从而保证不同类型 ETL 工作的效率。除了上述这些工作以外, 其他研究则从提高连接算法性能入手, 保证实时数据集成的效率和时效性。

连接操作直接关系到实时数据集成的性能, 因此, 高效的针对数据流的连接算法是保证实时数据集成的关键。早期的研究都是针对磁盘关系的高效连接技术, 比如索引嵌套循环连接, 归并连接和散列连接[9]。随着企业对实时数据需求的不断增加, 数据流更新的场景越来越多, 针对数据流更新的连接算法研究也引起越来越多研究人员的关注[10−12]。早期的一些研究工作[13−14]采用对称哈希连接来处理多路连接操作, 从而有效处理针对多个无边界数据流的连接查询操作。文献[14]提出的 Psoup 方法不仅允许新的查询使用旧的数据, 而且还允许就的查询使用新的数据。但是, 这些方法都假定使用基于时间的窗口或者基于元组的窗口, 对数据流进行分割, 从而使得数据流转变为许多可以放入内存的有限元组集。很显然, 这种假设在本文要解决的问题中是无法满足的, 在此问题中, 内存大小远小于关系 R 的大小, 关系 R 无法一次性放入内存, 同时, 也不存在针对数据流 S 的窗口限制, 即不会把数据流分成多个有限元组集。另外一些工作, 研究了针对数据流化的、有边界的关系的连接操作, 比如渐进合并连接 PMJ(progressive merge join)[15], 以及基于速率的渐进连接 RPJ(rate-based progressive join)[16]。这些方法可以对数据流输入进行连续地访问, 并且把接收到的元组放置到内存中, 从而以尽可能快的速度产生结果, 当接收到的元组超过内存容量限制时, 算法就把一部分元组刷新到磁盘中, 进行暂时存储, 当内存大小允许时, 再次把这些元组调入内存。很显然, 这种方法也是无法满足本文所研究问题的环境设置的, 因为, 针对数据流 S 的缓存会阻塞数据流更新操作, 无法满足实时更新的要求。

7 结论

实时数据仓库具有广阔的应用前景, 实时数据集成是实时数据仓库在实施过程中需要解决的关键问题。连接操作是数据集成中的典型操作, 通过设计高性能的连接算法来保证数据集成的实时性, 值得深入研究。

本文提出了 MESHJOIN 算法的改进算法, 即 MESHJOIN*算法。该算法可以减少原算法对当前连接无效元组的读取, 从而提高连接算法的效率。另外, MESHJOIN*算法采用了多线程并发连接技术, 并根据工程学原理, 实现了连接操作和关系 R 读取操作的最佳调度, 保证连接算法效率的最大化。MESHJOIN*算法对数据流更新元组的类型进行区分, 更新元组被分为实时元组和准实时元组, 优化器可以根据当前系统的服务率和数据流元组的到达率之间的关系, 合理调度不同类

型元组的执行，保证系统对实时元组的处理要求，并在此前提下尽可能提高系统整体处理效率。实验表明, MESHJOIN*算法可以取得比MESHJOIN算法更好的性能。

今后，将研究如何根据内存大小设计最优的关系 R 分块方法，从而保证连接算法取得更高的效率；同时，设计更加完善的关系 R 的块维护方法，使之在频繁更新的环境中，对系统整体影响最小化。

References:

[1] Bruckner B M, List B, Schiefer J. Striving towards near real-time data integration for data warehouses[C]//Kambayashi Y, Winiwarter W, Arikawa M. Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery. Berlin: Springer, 2002: 317–326.

[2] Araque F. Real-time data warehousing with temporal requirements[C]//Eder J, Mittermeir R, Pernici B. Proceedings of 2003 International Conference on Decision Systems Engineering, 2003: 293–304.

[3] Johnston-Bell D, Moore C, Garner B. Building a near real time active data warehouse[C]//Hamza M H. Proceedings of the International Conference on Databases and Applications. Calgary: IASTED/ACTA Press, 2004: 151–156.

[4] Lin Ziyu, Yang Dongqing, Song Guojie, et al. Materialized views selection of multi-dimensional data in real-time active data warehouses[J]. Chinese Journal of Software, 2008, 19 (2): 301–313.

[5] Polyzotis N, Skiadopoulos S, Vassiliadis P, et al. Supporting streaming updates in an active data warehouse[C]//Proceedings of the 23rd International Conference on Data Engineering. Istanbul: IEEE Computer Society Press, 2007: 476–485.

[6] Labio W J, Yerneni R, Garcia-Molina H. Shrinking the warehouse update window[J]. ACM SIGMOD Record, 1999, 28(2): 383–394.

[7] Kimball R. Real-time partitions[EB/OL]. [2010-05-07]. http://www.rkimball.com/html/designtipsPDF/DesignTips2001/KimballDT31Designing.pdf.

[8] Karakasidis A, Vassiliadis P, Pitoura E. ETL queues for active data warehousing[C]//Berti-Equille L, Batini C, Srivastava D. Proceedings of the International Workshop on Information Quality in Information Systems. New York: ACM Press, 2005: 28–39.

[9] Silberschatz A, Korth H F, Sudarshan S. Database system concepts[M]. 4th ed. New York: McGraw-Hill, 2001: 90–95.

[10] Babcock B, Babu S, Datar M, et al. Models and issues in data stream systems[C]//Popa L. Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. New York: ACM Press, 2002: 1–16.

[11] Babu S, Widom J. Continuous queries over data streams[J]. SIGMOD Record, 2001, 30(3): 109–120.

[12] Golab L, Ozsu M T. Issues in data stream management[J]. SIGMOD Record, 2003, 32(2): 5–14.

[13] Hammad M, Franklin M, Aref W, et al. Scheduling for shared window joins over data streams[C]//Proceedings of 29th International Conference on Very Large Data Bases. San Fransisco: Morgan Kaufmann, 2003: 297–308.

[14] Chandrasekaran S, Franklin M. PSoup: A system for streaming queries over streaming data[J]. The Journal of VLDB, 2003, 12(2): 140–156.

[15] Dittrich J P, Seeger B, Taylor D, et al. Progressive merge join: A generic and non-blocking sort-based join algorithm[C]//Proceedings of 28th International Conference on Very Large Data Bases. San Fransisco: Morgan Kaufmann, 2002: 299–310.

[16] Tao Y, Yiu M, Papadias D, et al. RPJ: Producing fast join results on streams through rate-based optimization[C]//Proceedings of the ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2005: 371–382.

附中文参考文献：

[4] 林子雨, 杨冬青, 宋国杰, 等. 实时主动数据仓库中多维数据实视图的选择[J]. 软件学报, 2008, 19(2): 301–313.

LIN Ziyu was born in 1978. He received his Ph.D. degree in Computer Software and Theory from Peking University in 2009. Now he is an assistant professor and M.S. supervisor at Xiamen University. His research interests include data warehouse, on-line analytical processing (OLAP), and data mining, etc.
林子雨(1978–)，男，吉林柳河人，2009 年于北京大学获得计算机软件与理论专业博士学位，目前为厦门大学计算机科学系助理教授、硕士生导师，主要研究领域为数据仓库，联机分析技术，数据挖掘等。

LIN Chen was born in 1982. She received her Ph.D. degree in Computer Software and Theory from Fudan University in 2010. Now she is an assistant professor and M.S. supervisor at Xiamen University. Her research interests include data retrieval, data mining and management, etc.
林琛(1982–)，女，福建厦门人，2010年于复旦大学获得计算机软件与理论专业博士学位，目前为厦门大学计算机科学系助理教授、硕士生导师，主要研究领域为数据检索，数据挖掘和管理等。

FENG Shaorong was born in 1964. He received his Ph.D. degree in Computer Application Technology from South China University of Technology in 2008. Now he is an associate professor and M.S. supervisor at Xiamen University. His research interests include data warehouse, on-line analytical processing (OLAP), and data mining, etc.
冯少荣(1964–)，男，河北南宫人，2008年于华南理工大学获得计算机应用技术专业博士学位，目前为厦门大学计算机科学系副教授、硕士生导师，主要研究领域为数据仓库，联机分析技术，数据挖掘等。

ZHANG Dongzhan was born in 1974. He received his Ph.D. degree in Computer Application Technology from Beijing Institute of Technology in 2009. Now he is an associate professor and M.S. supervisor at Xiamen University. His research interests include data warehouse, on-line analytical processing (OLAP), and data mining, etc.
张东站(1974–)，男，江苏徐州人，2009 年于北京理工大学获得计算机应用技术专业博士学位，目前为厦门大学计算机科学系副教授、硕士生导师，主要研究领域为数据仓库，联机分析技术，数据挖掘等。